EXHIBIT 9

Computer Communication Review

acm sigcomm

ACM PRESS

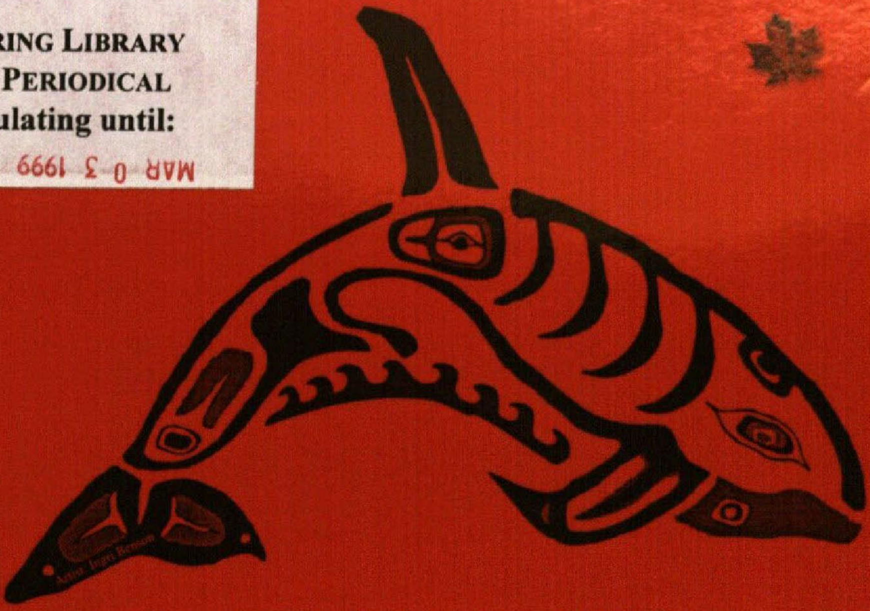
Special Interest Group on Data Communication

An ACM SIGCOMM Publication

Volume 28
Number 4
October 1998

Proceedings of

ACM SIGCOMM'98 CONFERENCE



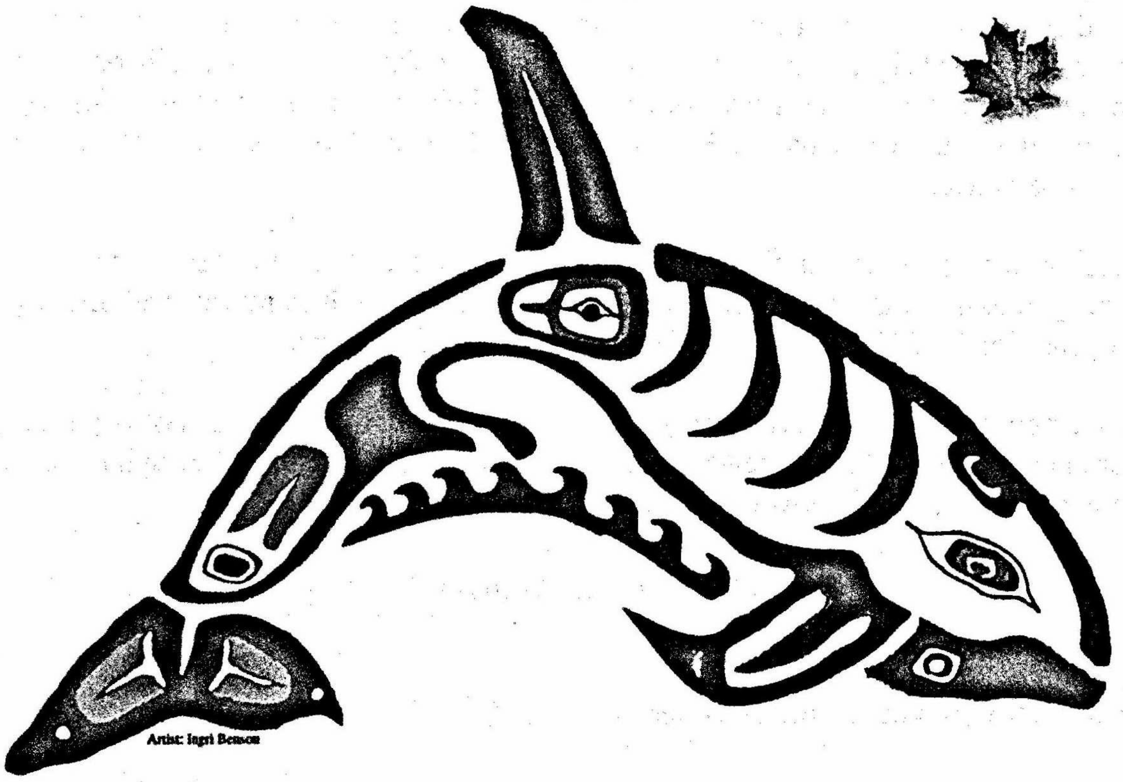
Applications, Technologies, Architectures, and Protocols for Computer Communication

Vancouver, British Columbia, Canada
August 31 to September 4, 1998

ACM

SIGCOMM'98

Proceedings



Applications, Technologies, Architectures, and Protocols for Computer Communication

http://www.acm.org/sigcomm/sigcomm98

Vancouver, British Columbia, Canada
September 2 to September 4, 1998
(Tutorials August 31 and September 1)

Bay Networks BC Bellcore HEWLETT PACKARD
IBM NEWBRIDGE PMC PMC-Sierra, Inc. Making Global Networks Work SIARA SYSTEMS

Router Plugins
A Software Architecture for Next Generation Routers

Dan Decasper[1], Zubin Dittia[2], Guru Parulkar[2], Bernhard Plattner[1]
[dan|plattner]@tik.ee.ethz.ch, [zubin|guru]@arl.wustl.edu
[1]Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland
Phone: +41-1-632 7019 Fax: +41-1-632 1035
[2]Applied Research Laboratory, Washington University, St. Louis, USA
Phone: +1-314-935 4586 Fax: +1-314-935 7302

1. ABSTRACT

Present day routers typically employ monolithic operating systems which are not easily upgradable and extensible. With the rapid rate of protocol development it is becoming increasingly important to dynamically upgrade router software in an incremental fashion. We have designed and implemented a high performance, modular, extended integrated services router software architecture in the NetBSD operating system kernel. This architecture allows code modules, called *plugins*, to be dynamically added and configured at run time. One of the novel features of our design is the ability to bind different plugins to individual flows; this allows for distinct plugin implementations to seamlessly coexist in the same runtime environment. High performance is achieved through a carefully designed modular architecture; an innovative packet classification algorithm that is both powerful and highly efficient; and by caching that exploits the flow-like characteristics of Internet traffic. Compared to a monolithic best-effort kernel, our implementation requires an average increase in packet processing overhead of only 8%, or 500 cycles/2.1ms per packet when running on a P6/233.

1.1 Keywords

High performance integrated services routing, modular router architecture, router plugins

2. INTRODUCTION

New network protocols and extensions to existing protocols are being deployed on the Internet. New functionality is being added to modern IP routers at an increasingly rapid pace. In the past, the main task of a router was to simply forward packets based on a destination address lookup. Modern routers, however, incorporate several new services:

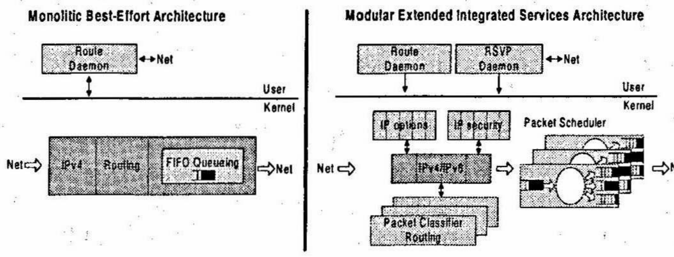


Figure 1. : Best Effort vs Extended Integrated Services Router (EISR)

- Integrated/differentiated Services
- Enhanced routing functionality (level 3 and level 4 routing and switching, QoS routing, multicast)
- Security algorithms (e.g. to implement virtual private networks (VPN))
- Enhancements to existing protocols (e.g. Random Early Detection (RED))
- New core protocols (e.g. IPv6 [8])

Figure 1 contrasts the software architecture of our proposed Extended Integrated Services Router (EISR) with that of a conventional best-effort router. A typical EISR kernel features the following important additional components: a packet scheduler, a packet classifier, security mechanisms, and QoS-based routing/Level 4 switching. Various algorithms and implementations of each component offer specific advantages in terms of performance, feature sets, and cost. Most of these algorithms undergo a constant evolution and are replaced and upgraded frequently. Such networking subsystem components are characterized by a relatively "fluid" implementation, and should be distinguished from the small part of the network subsystem code that remains relatively stable. The stable part (called the core) is mainly responsible for interacting with the network hardware and for demultiplexing packets to specific modules. Different implementations of the EISR components outside of the core often need to coexist. For example, we might want to use one kind of packet scheduling on one interface, and a different kind on another.

In this paper, we propose a software architecture and present an implementation which addresses these requirements. The specific goals of our framework are:

- **Modularity**: Implementation of specific algorithms come in the form of modules called *plugins*[1].

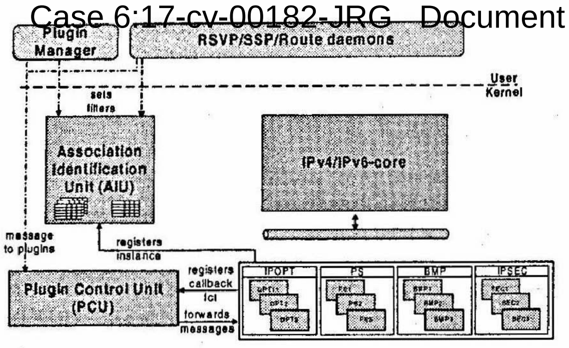


Figure 2. : System Architecture and Control Path

- **Plugins:** Figure 2 shows four different types of plugins - plugins implementing IPv6 options, plugins for packet scheduling, plugins to calculate the best-matching prefix (BMP, used for packet classification and routing), and plugins for IP security. Other plugin types are also possible: e.g., a routing plugin, a statistics gathering plugin for network management applications, a plugin for congestion control (RED), a plugin monitoring TCP congestion backoff behaviour, a firewall plugin. Note that all plugins come in the form of dynamically loadable kernel modules.
- **Plugin Control Unit (PCU):** The PCU manages plugins, and is responsible for forwarding messages to individual plugins from other kernel components, as well as from user space programs (using library calls).
- **Association Identification Unit:** The Association Identification Unit (AIU) implements a packet classifier and builds the glue between the flows and plugin instances. The operation of the AIU will become clear when we describe the data path in the next subsection.
- **Plugin Manager:** The Plugin Manager is a user space utility used to configure the system. It is a simple application which takes arguments from the command line and translates them into calls to the user-space *Router Plugin Library* which we provide with our system. This library implements the function calls needed to configure all kernel level components. In most cases, the plugin manager is invoked from a configuration script during system initialization, but it can also be used to manually issue commands to various plugins. We show an example of how the Plugin Manager is used in Section 6.
- **Daemons:** The RSVP [31], SSP [1] (a simplified version of RSVP), and route daemon are linked against the Router Plugin Library to perform their respective tasks. We implemented an SSP daemon for our system, and are currently in the process of porting an RSVP implementation.

After a reboot, the system has to be configured before it is ready to receive and forward data packets. Configuration involves the selection of a set of plugins. Since a selection does not necessarily apply to all packets traversing the router, a definition of the set of packets which should be processed by each individual plugin instance is required. This configuration can be done either by a system administrator, or by executing a script. Configuration involves the following steps:

- **Loading a plugin:** Using the *modload* command, which is part of the NetBSD distribution, plugins are loaded into the kernel. On loading, they register themselves with the PCU by providing a callback function. This function is used to send messages to the plugin. There are messages for creating and freeing instances of the plugin and for binding plugin instances to flows. Also, plugin developers can define an arbitrary number of plugin specific messages. Once the callback function for a plugin has been registered, the PCU can forward these configuration messages to the plugin.
- **Creating an instance of a plugin:** Using the Plugin Manager application, configuration messages can be sent to specified plugins. Typically, these messages ask the plugin to create an instance of itself. In case of a packet scheduling plugin for example, the configuration information could include the network interface the plugin should work on.
- **Creating filters:** Once a plugin has been configured and an instance has been created, it is ready to be used. What has to be defined next is the set of datagrams which should be passed to the instance for processing. This is done by binding one or more flows to the plugin instance. To specify the set of flows that are supposed to be handled by a particular plugin instance, the Plugin Manager or one of the user space daemons (RSVP or SSP) can create filters through calls to the AIU. Recall (from earlier in this section) that a filter is a specification for the set of flows it matches.
- **Binding flows to instances:** Next, the binding between filters and plugin instances must be established. Each filter in the AIU is associated with a pointer to a plugin instance; this pointer is set by making another call to the AIU to do the binding.

Now the system is ready to process data packets. We will show in the next subsection how data packets are matched against filters and how they get passed to the appropriate instances.

3.2 The Data Path

Data packets in our system are passed to instances of plugins which implement the specific functions for processing the packets. Since data path mechanisms are applied to every single packet, it is very important to optimize their performance. Given a packet, our architecture should be able to quickly and efficiently discover the set of instances that will act on the packet.

The data path interactions are shown in Figure 3.Before we can explain the sequence of actions, we have to introduce the notion of a gate.

A ***gate*** is a point in the IP core where the flow of execution branches off to an instance of a plugin. From an implementation point of view, gates are simple macros which encapsulate function calls to the AIU that will return